React Redux Advance :

In react redux advance we are going to test the components one by one component by component.

There are 3 types of testing :

1. Static testing:

* In this type of testing we render the given component and return a plain HTML, such like :

We have the (it) statement which we use it as a testing tool such like :

it('shows a comment box', ()=>{

here it’s like we are reading the statement locally like the component shows a “comment box” statement

(it shows a comment box)

And the second attribute is an arrowed function which includes :

ReactDOM.render(<App />, div);

The component tag and the name of the element inside the component

We can write this statement to test the component’s element

expect(div.innerHTML).toContain('Comment Box');

where :

* Expect : (Expectation) and it’s a global function
* (div.innerHTML) : the value that we are inspecting/ thing we want to verify
* .toContain : called the matcher statement it clarify how we trying to inspect the thing that we passed it in the previous statement
* (‘comment box’) : the value that we expect to see

But we can use this instead of the previously code:

ReactDOM.unmountComponentAtNode(div);

1. In this type we use enzyme :

* Import a specific function from enzyme :

import {shallow} from 'enzyme';

* Then we write this :

const wrapped = shallow(<App />);

* We are using the terminology (wrapped) to indicate that the object that we got back from ( shallow(<App />) ) is a wrapped version of our App component
* Wrapped means that this is a component that has some additional functionality loaded on top
* Then we import the component that we want to test it which is here the CommentBox component
* Now we can write our expectation of this component by :

expect(wrapped.find(CommentBox));

//inside of expect is the value that we are trying

//to ma assertion about or expectation about

The find function is going to return an array, this array will return every instance of CommentBox been found

expect(wrapped.find(CommentBox).length).toEqual(1);

here it will check the number of instances of CommentBox and we are expecting to have (1) instance so we use (toEqual) function to test if we will get a result of (1).

**Note:**

Now instead of writing a huge number of imports for all of the components we will use the (Absolute Path Imports) and this will be to the src directory :

1. By making a new file in the main project which called (testing)
2. Call the file ( .env )
3. Write inside it NODE\_PATH = src/
4. And then go to App.test.js and refactor the import statements

* Now we want to use a function before every test so we can add more logic to our test and it’s provided by JEST:

let wrapped;

beforeEach (()=>{

wrapped = shallow(<App />);

});

it('shows a comment box', ()=>{

expect(wrapped.find(CommentBox).length).toEqual(1);

//inside of expect is the value that we are trying

//to ma assertion about or expectation about

});

it('shows a comment list', ()=>{

expect(wrapped.find(CommentList).length).toEqual(1);

//inside of expect is the value that we are trying

//to ma assertion about or expectation about

});

* Now we will move to CommentBox component:
* Replace the functional component with a class based component because it’s going to be responsible for watching input from the user so it must be a class based component
* Now we made some elements inside the CommentBox component:

We made a simple form with ( h4, textarea, div🡪 button )

* And then we want to create a new piece of state that’s gonna be managed from the textarea, so it’s gonna be turned to a controlled input
* Now we initialize an initial value for the state by writing above the render method :

State = { comment: “ ” }

* And then we assign it to text area :

< textarea value= {this.state.comment} />

* Then we must make a new function for the textarea to enable it to change its content or lets say to write a comment inside it :

1. Put onChange = {this.handleChange} inside textarea
2. Declare a fat arrowed function that receives an even as an argument
3. Then inside of handleChange function we say setState=({comment: event.target.value})

* And then we make an event handler on the form so when we submit it , it will take the comment and clear the textarea, also it will prevent the browser from refreshing:

(we will call it “handleSubmit” and put it inside the form tag)

* **NOW** we will test the CommentBox component:

1. We will make sure that we have a visible textarea and a button that the user can interact with.
2. We will make sure that the user can enter some text into that textarea and submit it.
3. We will make sure that anytime the input got submitted the textarea itself should be emptied.

* We need to decide which of the 3 enzyme test handlers we should choose.

The CommentBox doesn’t has any child underneath it so we will choose the shallow test handler (and that is the logical way)

But for more experience Stephen decided to choose the Full DOM way.

1. We will create a new file to do this test **(CommentBox.test.js)**

* Import react and the component itself
* **Here we are making sure that we have a visible textarea and a button that the user can interact with.(step #1)**
* Import the full DOM test by { mount }🡪import {mount} from ‘enzyme’;
* Create the first (it) statement
* Then apply the expect statement

it('has a text area and a button', () => {

expect(wrapped.find('textarea').length).toEqual(1);

expect(wrapped.find('button').length).toEqual(1);

});

* Declare wrapped as (let) not (const) and put it outside the (it) statement.
* Declare a beforeEach function and put the wrapped inside it to use it globally

beforeEach(()=>{

wrapped = mount(<CommentBox />);

})

* And now we will meet the afterEach helping function

It’s the **exact opposite** of the beforeEach helping function

It will run a function or a little chuck of code that I pass it after every single test executed.

beforeEach(()=>{

wrapped = mount(<CommentBox />);

});

afterEach(()=>{

wrapped.unmount();

});

it('has a text area and a button', () => {

expect(wrapped.find('textarea').length).toEqual(1);

expect(wrapped.find('button').length).toEqual(1);

});

This is the afterEach helping function:

* Inside it we use the unmount function to clean up after the testing operation
* The way it works is so easy:

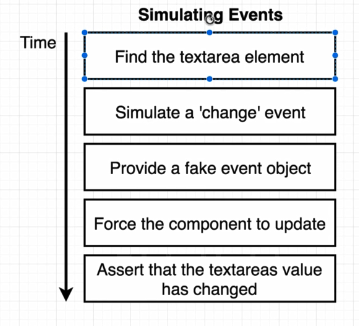
beforeEach executed

then (it) statement

after them the afterEach executed to clean up

* **Here we are making sure that the user can enter some text into that textarea and submit it. (step #2)**

This is the flow :



**3**

**4**

**5**

**2**

**1**

1. It’s very similar to the test that we wrote it in step #1 by using

( .find ) function

We declare a new (it) statement and then:

it('has a text area that the user can type in',()=>{

wrapped.find('textarea');

});

1. When we simulate our change event it will automatically call our callback function of the handleChange callback

it('has a text area that the user can type in',()=>{

wrapped.find('textarea').simulate('change');

});

When we simulate a component we need to use an html real names not a react names from (onChange) we take just the name of the event so we choose (change)

1. That will allow us to customize our event object and kind of trick our component to think that its just called with a certain value for this new textarea that the user just typed in

it('has a text area that the user can type in',()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

});

});

Here we are using the mock object as a second argument of the simulation operation and in it we are trying to give a mock value to the text area as an object that has another object that equals the value of the state

1. What is the reason of this step ??

The reason of it is to re-render the setState call every time we need to change the state so we going to **force** our component to re-render and a new value is showed so we can then right an assertion

it('has a text area that the user can type in',()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

});

wrapped.update();

});

And that happened by writing ( wrapped.update() ).

it('has a text area that the user can type in',()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

});

wrapped.update();

expect (wrapped.find('textarea').prop('value')).toEqual('new comment'); });

* **Here we will make sure that anytime the input got submitted the textarea itself should be emptied.**

1. Make a new (it) statement

it('when form is submitted, text area gets emptied',()=>{

});

1. We want to make sure that the textarea is taking a right comment

it('when form is submitted, text area gets emptied',()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

})

wrapped.update();

});

1. Now we want to find the form to test the submit on it and then update it

it('when form is submitted, text area gets emptied',()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

})

wrapped.update();

wrapped.find('form').simulate('submit');

wrapped.update();

});

1. After the form is submitted absolutely it will make the textarea emptied so we test it like this

it('when form is submitted, text area gets emptied', ()=>{

wrapped.find('textarea').simulate('change',{

target: {value: 'new comment'}

})

wrapped.update();

wrapped.find('form').simulate('submit');

wrapped.update();

expect(wrapped.find('textarea').prop('value')).toEqual('');

});